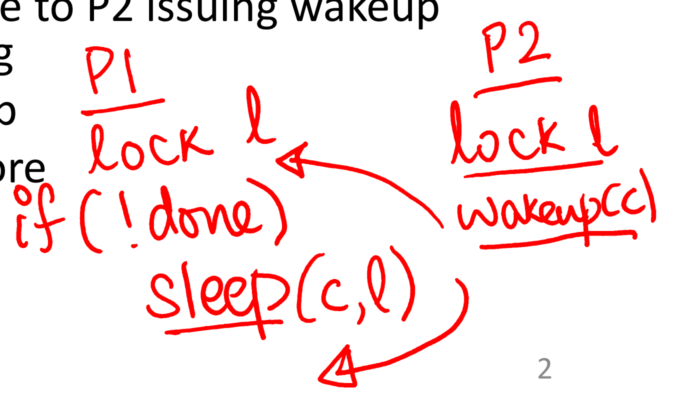
Lecture 30: Sleep and wakeup in xv6

Mythili Vutukuru
IIT Bombay
https://www.cse.iitb.ac.in/~mythili/os/

Sleep and wakeup

- A process P1 in kernel mode gives up CPU to block on a event
  - Example: process reads a block from disk, must block until disk read completes
- P1 invokes "sleep" function, which calls sched() and gives up CPU
- Another process P2 in kernel mode calls "wakeup" when event occurs, marks P1 as runnable, scheduler loop switches in P1 in future
  - Example: disk interrupt occurred when P2 is running, so P2 handles the interrupt, and marks P1 as runnable
- How does P2 know which process to wake up? When P1 sleeps, it sets a channel (void * chan) in its struct proc, and P2 calls wakeup on same channel (channel = any value known to both P1 and P2)
  - Example: channel value for disk read can be address of disk block
- Spinlock protects atomicity of sleep: P1 calls sleep with some spinlock L held, P2 calls wakeup with same spinlock L held
  - Eliminating missed wakeup problem that arises due to P2 issuing wakeup between P1 deciding to sleep and actually sleeping
  - Lock L released after sleeping, available for wakeup
  - Similar concept to condition variables studied before



Sleep function

- Sleep calls sched() to give up CPU
  - Needs to hold ptable.lock
- Acquire ptable.lock, release the lock given to sleep (make it available for wakeup)
  - Unless lock given is ptable.lock itself, in which case no need to acquire again
  - One of two locks held at all times
- Calls sched(), switched out of CPU, resumes again when woken up and ready to run
- Reacquires the lock given to sleep and returns back
  - Code that invoked sleep with lock held returns with lock held again

```
2871 // Atomically release lock and sleep on chan.
2872 // Reacquires lock when awakened.
2873 void
2874 sleep(void *chan, struct spinlock *lk)
2875 {
2876   struct proc *p = myproc();
2877
2878   if(p == 0)
2879     panic("sleep");
2880
2881   if(lk == 0)
2882     panic("sleep without lk");
2883
2884   // Must acquire ptable.lock in order to
2885   // change p->state and then call sched.
2886   // Once we hold ptable.lock, we can be
2887   // guaranteed that we won't miss any wakeup
2888   // (wakeup runs with ptable.lock locked),
2889   // so it's okay to release lk.
2890   if(lk != &ptable.lock){
2891     acquire(&ptable.lock);
2892     release(lk);
2893   }
2894   // Go to sleep.
2895   p->chan = chan;
2896   p->state = SLEEPING;
2897
2898   sched();
2899
2900   // Tidy up.
2901   p->chan = 0;
2902
2903   // Reacquire original lock.
2904   if(lk != &ptable.lock){
2905     release(&ptable.lock);
2906     acquire(lk);
2907   }
2908 }
```

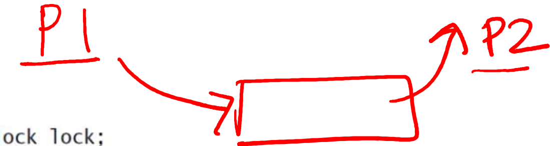
ptable.lock

Wakeup function

- Called by another process with lock held (same lock as when sleep was called)
- Since it changes ptable, ptable.lock will also be held
  - If sleep lock is ptable.lock itself, then directly call wakeup1
- Sleep holds one of sleep's lock or ptable.lock at all times, so a wakeup cannot run in between sleep
- Wakes up all processes sleeping on a channel in ptable (more like signal broadcast of condition variables)
  - Good idea to check condition is still true upon waking up (use while loop while calling sleep)

```
2950 // Wake up all processes sleeping on chan.
2951 // The ptable lock must be held.
2952 static void
2953 wakeup1(void *chan)
2954 {
2955   struct proc *p;
2956
2957   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2958     if(p->state == SLEEPING && p->chan == chan)
2959       p->state = RUNNABLE;
2960 }
2961
2962 // Wake up all processes sleeping on chan.
2963 void
2964 wakeup(void *chan)
2965 {
2966   acquire(&ptable.lock);
2967   wakeup1(chan);
2968   release(&ptable.lock);
2969 }
```

lock l
wakeup
release l




```
6762 struct pipe {
6763   struct spinlock lock;
6764   char data[PIPESIZE];
6765   uint nread;      // number of bytes read
6766   uint nwrite;     // number of bytes written
6767   int readopen;    // read fd is still open
6768   int writeopen;   // write fd is still open
6769 };
```

Example: pipes

- Two processes connected by a pipe (producer consumer)
  - Common shared buffer, protected by a spinlock
- One process writes into pipe, another reads from pipe
- Reader sleeps if pipe is empty, writer wakes it up after putting data
- Writer sleeps when pipe is full, reader wakes it up when data is consumed
- Addresses of pipe structure variables are channels (same channel known to both)

```
6829 int
6830 pipewrite(struct pipe *p, char *addr, int n)
6831 {
6832   int i;
6833
6834   acquire(&p->lock);
6835   for(i = 0; i < n; i++){
6836     while(p->nwrite == p->nread + PIPESIZE){
6837       if(p->readopen == 0 || myproc()->killed){
6838         release(&p->lock);
6839         return -1;
6840       }
6841       wakeup(&p->nread);
6842       sleep(&p->nwrite, &p->lock);
6843     }
6844     p->data[p->nwrite++ % PIPESIZE] = addr[i];
6845   }
6846   wakeup(&p->nread);
6847   release(&p->lock);
6848   return n;
6849 }
```

```
6850 int
6851 piperead(struct pipe *p, char *addr, int n)
6852 {
6853   int i;
6854
6855   acquire(&p->lock);
6856   while(p->nread == p->nwrite && p->writeopen){
6857     if(myproc()->killed){
6858       release(&p->lock);
6859       return -1;
6860     }
6861     sleep(&p->nread, &p->lock);
6862   }
6863   for(i = 0; i < n; i++){
6864     if(p->nread == p->nwrite)
6865       break;
6866     addr[i] = p->data[p->nread++ % PIPESIZE];
6867   }
6868   wakeup(&p->nwrite);
6869   release(&p->lock);
6870   return i;
6871 }
```

Example: wait and exit

- If wait called in parent while children are still running, parent calls sleep and gives up CPU
  - Here, channel is parent struct proc pointer, lock is ptable.lock

```
2706        // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2707        sleep(curproc, &ptable.lock);
```

- In exit, child acquires ptable.lock and wakes up sleeping parent

```
2650    // Parent might be sleeping in wait().
2651    wakeup1(curproc->parent);
```

- Here, lock given to sleep is ptable.lock because parent and child both access ptable (sleep avoids double locking, doesn't acquire ptable.lock if it is already held before calling sleep)
- Why is terminated process memory cleaned up by parent? When a process calls exit, CPU is using its memory (kernel stack is in use, cr3 is pointing to page table) so all this memory cannot be cleared until terminated process has been taken off the CPU
  - Parent code in wait is a good place to clean up child memory after child has stopped running

Summary

- Sleep and wakeup functionality in kernel for processes to wait for or signal each other
  - Similar to condition variables for synchronization of user space threads
- Examples of sleep/wakeup
  - Pipe reader and pipe writer processes
  - Parent sleeps for child to die, zombie child wakes up parent
- Code calling sleep and wakeup need to hold same lock, in order to avoid missed wakeup problem